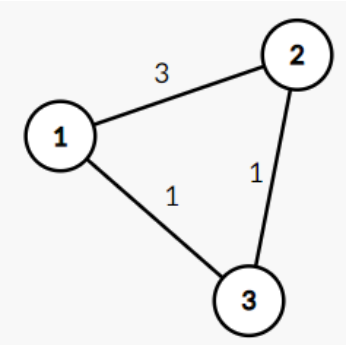
Algoritmi pentru determinarea drumurilor minime în grafuri.

Roy-Floyd, **Bellman-Ford**, **Dijkstra** - *prezentare pe scurt și analiză comparativă*.

Prezentările de mai sus sunt valabile atât pentru grafuri neorientate cât și pentru cele orientate.

Dacă muchiile nu au asociate costuri, determinarea drumurilor minime se realizează și cu o parcurgere în lățime (`BFS`).

În cazul grafului cu costuri pe muchii `BFS` nu mai oferă soluția problemei. Iată un exemplu prin care justificăm acest lucru:



Aplicând în acest graf `BFS` cu sursă nodul `1`, punem imediat în coadă pe `2` și `3` ca vecini ai lui `1` (și totodată se marchează `1` ca fiind tata pentru `2` și pentru `3` în arborele de acoperire). Astfel, ca drum minim de la `1` la `2` s-ar găsi ca fiind chiar muchia (`1`,`2`), de cost `3`, însă un drum mai bun, de cost `2`, ar fi fost `1`, `3`, `2`.

În concursurile de programare se folosesc de de cele mai multe ori cei trei algoritmi enumerați în titlu.

Prezentăm mai întâi pe scurt pe fiecare dintre ei apoi vom face o analiză.

Roy-Floyd

Este cel mai simplu de scris și de învățat, dar nu și cel mai eficient pe cazul general.

Se pornește de la matricea costurilor:

`C[i][j]` reprezintă costul muchiei de la `i` la `j` (dacă există muchie) sau un marcaj (dacă nu există muchie). În cazul lipsei muchiei acest marcaj se pune o valoare foarte mare, să îi spunem `INF`.

Matricea costurilor este un fel de matrice de adiacență în care în loc de `1` punem costul muchiei și în loc de `0` punem `INF`.

Vom modifica matricea pas cu pas și la final vom obține `C[i][j]` = *distanța minimă de la `i` la `j`*.

Matricea `C` o vom actualiza în `n` etape:

- Inițial `C[i][j]` reprezintă chiar lungimea muchiei directe de la `i` la `j` (sau `INF` dacă nu e muchie).
- Introducem apoi ca intermediar nodul `1` și vom transforma `C` astfel încât `C[i][j]` = lungimea minimă a unui drum de la `i` la `j` folosind muchia directă sau nodul intermediar `1`.
- Introducem apoi ca intermediar pe `2` și, pornind de la matricea obținută la pasul anterior, vom transforma `C` așa încât `C[i][j]` să reprezinte drumul minim de la `i` la `j` folosind muchia directă și intermediarii `1` și `2`.
- Introducând pe rând fiecare nod ca intermediar, obținem la final matricea `C` ca fiind **matricea drumurilor minime**.

Observăm că acest algoritm permite să obținem distanța minimă între oricare două noduri ale grafului. Complexitatea în timp a sa este de ordinul n^3 întrucât avem `n` pași (câte unul pentru fiecare intermediar) și la fiecare pas avem de analizat fiecare element al matricei.

La un pas, cu intermediarul `k`, presupunem că avem în `C[i][j]` distanța minimă de la `i` la `j` folosind muchia directă și intermediarii `1`, `2`, …, `k-1`.

Codul următor face ca la `C[i][j]` să se ia în calcul și intermediarul `k`.

```
for (i=1;i<=n;i++)
    for (j=1;j<=n;j++)
        if (C[i][j] > C[i][k] + C[k][j])
            C[i][j] = C[i][k] + C[k][j];
```

Variem `k` între `1` și `n`, așa cum am descris și obținem algoritmul:

```
for (k=1;k<=n;k++)
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            if (C[i][j] > C[i][k] + C[k][j])
                C[i][j] = C[i][k] + C[k][j];
```

Iată mai departe descrierea unui alt algoritm care se obține din cel descris anterior.

Considerăm `C` ca fiind inițial matricea de **adiacență**.

`C[i][j]` = `1` dacă am muchie și `0` dacă nu.

```
for (k=1;k<=n;k++)
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            if (C[i][k] == 1 && C[k][j] == 1)
                C[i][j] = 1;
```

În final: `C[i][j] = 1` dacă și numai dacă există drum de la `i` la `j`.

Matricea `C` astfel construită se mai numește **matricea drumurilor** iar algoritmul de mai sus se cheamă **Roy-Walshall**.

Bellman Ford

Dacă algoritmul Roy-Floyd determină drumul minim între oricare două noduri ale grafului, următorii doi permit să obținem drumul minim între un nod dat, de pornire, și toate celelalte.

Memorăm graful folosind liste de vecini.

Notăm cu `start` nodul de pornire (deci vom obține drumul minim între `start` și toate celelalte).

Folosim un vector `D` în care la final vom avea: `D[i]` = lungimea drumului minim de la start la nodul `i`.

Inițial, `D[start]` = `0` și `D[i]` = `INF`, pentru `i!=start`

Ideea de bază, este următoarea:

- Se parcurge lista de muchii și dacă pentru vreuna distanța minimă la extremitatea finală se actualizează din distanta minimă la extremitatea inițială plus costul muchiei, se operează în `D`.
- Dacă la o parcurgere s-a făcut cel puțin o actualizare în `D`, trebuie reluată parcurgerea muchiilor (un principiu simular cu acela de la bubble sort).

```
Inițial: D[start] = 0, în rest D[i] = INF
for (;;) {
    modificari = 0;
    pentru fiecare muchie (x->y)
        if (D[y] > D[x] + cost(x,y)) {
            D[y] = D[x] + cost(x,y)
            modificari = 1;
        }
    if (modificari == 0)
        break;
}
```

Optimizarea codului anterior presupune să păstrăm într-o coadă doar nodurile la care a fost modificată valoarea `D` și la care nu au fost analizați pentru optimizare vecinii. Acest lucru face să analizăm doar muchii care pot produce actualizări în `D`, nu neapărat tot setul de muchii de fiecare dată.

Aceasta face ca implementarea algoritmul Bellman Ford să aducă foarte mult cu BFS.

Plasăm mai întâi în coadă nodul de pornire.

În mod repetat:

Extragem primul nod al cozii (să îi spunem front) și analizăm vecinii săi.

Dacă D[vecin] > D[front] + (cost muchie front-vecin)

D[vecin] = D[front] + (cost muchie front-vecin);

Aici un detaliu de implementare important:

Nodurile care se află în coadă la un moment dat se țin marcare într-un vector de frecvență (să îi spunem `viz`. Deci inițial `viz[start] = 1`).

Când extragem nodul front din coadă facem `viz[front] = 0`

Însă, dacă actualizăm valoarea `D` pentru un nod vecin, ca mai sus, va trebui să ne asigurăm că acest vecin va ajunge în coadă (practic aceasta este ideea de bază a algoritmului: dacă unui nod i s-a modificat valoarea distanței minime de la sursă, prin intermediul lui se va putea modifica valoarea distanței minime de la `start` la vecinii săi, deci trebuie să ajungă în coadă pentru a-l putea expanda).

Este posibil ca nodul vecin, cu valoarea din `D` tocmai modificată, să fie deja în coadă (pentru asta se verifică valoarea `viz[vecin]`), și în acest caz nu îl mai introducem. În caz contrar punem pe vecin în coadă și facem `viz[vecin] = 1`.

Când coada se va goli, înseamnă că nu se mai fac actualizări de distanță minimă de la `start` la vreun nod, și algoritmul se oprește.

Pe scurt, algoritmul este următorul:

```
q.push_back(start);
viz[start] = 1; /// viz indica daca momentan nodul e in coada
while (!q.empty()) {
    front = q.front();
    q.pop_front();
    viz[front] = 0;
    for (vecin al lui front)
        if (D[vecin] > D[front] + cost_muchie(front, vecin) {
            D[vecin] = D[front] + cost_muchie(front, vecin);
            if (viz[vecin] == 0) {
                q.push_back(vecin);
                viz[vecin] = 1;
            }
        }
}
```

Semnificația elementelor din coadă este că acelor noduri le-a fost actualizat `D` dar noua valoare nu este încă luată în calcul la valoarea din `D` a vecinilor lor.

Complexitatea acestui algoritm tinde către n^3 (observăm că un nod poate ajunge în coadă de mai multe ori, spre deosebire de BFS) însă în practică se comportă foarte bine, fiind deseori o alternativă la Dijkstra, scriindu-se și mai repede.

O altă facilitate a acestui algoritm este aceea că îl putem utiliza la detectarea ciclilor de cost negativ. Adică, în unele cazuri muchiile pot avea costuri negative. Dacă ajungem pe un ciclu cu suma costurilor muchiilor negativă, acest algoritm va rula infinit, tot scăznd costul drumului minim la nodurile de pe ciclu. Din fericire, cu o ușoară modificare putem detecta un astfel de ciclu.

Faptul că un nod ajunge să fie pus în coadă de mai mult de n ori implică existența unui ciclu de cost negativ. Pentru detectare putem modifica codul anterior astfel:

```
q.push_back(start);
viz[start] = 1;
iq[start] = 1; // iq = "in queue"
while (!q.empty()) {
    front = q.front();
    q.pop_front();
    viz[front] = 0;
```

```
    for (vecin al lui front)
        if (D[vecin] > D[front] + cost_muchie(front, vecin) {
            D[vecin] = D[front] + cost_muchie(front, vecin);
            if (viz[vecin] == 0) {
                iq[vecin]++;
                if (iq[vecin] > n) {
                     Avem ciclu de cost negativ și oprim algoritmul
                }
                q.push(back(vecin);
                viz[vecin] = 1;
            }
        }
}
```

Dijkstra

Ca și Bellman Ford, folosește un vector `D` cu aceeași semnificație:

`D[i]` = lungimea drumului minim de la start la nodul `i`.

Inițial, `D[start]` = `0` și `D[i]` = `INF`, pentru `i!=start`.

În cazul acestui algoritm facem, de asemenea, `n` pași. La fiecare pas extragem încă un nod la care am obținut minimul.

Astfel, la un moment dat, vom avea o mulțime de noduri la care am obținut minimul și mulțimea complementară, a nodurilor la care nu am obținut încă minimul. Facem distincție între cele două printr-un vector de frecvență, `viz`.

`viz[i]` = `1` dacă am obținut deja drumul minim de la start la `i` și `0` în caz contrar.

Inițial toate elementele din `viz` sunt `0`.

La un pas dintre cei `n`:

- Alegem poziția `p` cu `D[p]` minim și `viz[p]` = `0`. Practic acesta este noul nod la care am determinat minimul.
- Marcăm de acum acest nod în `viz` (`viz[p]` = `1`) și parcurgem vecinii săi pentru a actualiza valoarea `D` a lor.

Este o strategie greedy, în care, după ce repetăm cei doi pași de `n` ori vom obține în `D` valorile minime dorire.

Pe scurt, algoritmul este:

```
for (i=1;i<=n;i++)
    D[i] = INF;
D[start] = 0;
for (i=1;i<=n;i++){
    minim = INF;
    p = -1;
    for (j=1;j<=n;j++)
        if (viz[j] == 0 && D[j] < minim) {
            minim = viz[j];
            p = j;
        }

    if (minim == INF)
        break;
    viz[p] = 1;

    for (vecin al lui p)
        if (D[vecin] > D[p] + cost_muchie(p, vecin)) {
            D[vecin] = D[p] + cost_muchie(p, vecin);
            /// T[vecin] = p;
        }
}
```

Putem spune și altfel: la un moment dat, pentru nodurile marcate cu `1` în `viz` avem în `D` chiar lungimea drumului minim, iar pentru nodurile celelalte avem lungimea drumului minim care folosește ca intermediari doar noduri marcate.

De asemenea, dacă ținem cont de linia comentată mai sus, `T[vecin] = p;` ne putem imagina că nodurile marcare la un moment dat cu `1` în `viz` formează un arbore parțial (cu rădăcina în `start`) al drumurilor minime de la `start` la acele noduri. Pentru celelalte noduri, în acel moment se ține în `D` distanța minimă de la `start`, folosind ca intermediari doar nodurile din arbore.

Este ușor de observat că acest algoritm greedy are complexitatea n^2.

O altă observație este aceea că dacă nu se poate ajunge de la start la toate nodurile grafului, algoritmul se va opri după mai puțin de `n` etape (`break`).

Dijkstra admite totuși o optimizare care îl face un algoritm și mai bun:

Întrucât `D` este folosit pentru operații de extragere a minimului și actualizare a elementelor, el poate fi organizat ca un **heap**. Astfel primul for se transformă într-o instrucțiune de obținere a vârfului heap-ului: `o(1)`.

Al doilea for însă are nevoie de inserare a unei valori în heap și aduce un factor logaritmic pentru acest lucru.

Practic timpul de calcul ajunge la `n + m log n` (`log`-ul se aplică cel mult o dată la fiecare muchie).

Mai este un detaliu subtil ce ține de actualizarea valorii din vecini (al doilea `for`). Dacă se folosește structura preimplementată **priority_queue** pentru a ține heapul, noi facem inserări în ea, dar acestea trebuie să fie perechi (`cost`, `nod`). Dacă se folosește pair, se pune costul ca prim element al perechii, pentru că el trebuie se ia în calcul primul la comparare. Astfel, pentru un nod rămâne în heap și perechea corespunzătoare valorii costului să anterior și crește foarte mult necesarul de memorie. Pe de altă parte, ștergerea elementului din heap are complexitate, din păcate, liniară.

Alternativele în practică sunt: să se implementeze heap-ul de mână sau să se folosească structura **set**, în care se face mai întâi operația de ștergere a vechii perechi, și apoi inserarea noii perechi, ambele cu timp de calcul logaritmic.

În codul de mai jos avem `start` **=** `1`

```
for (i=1;i<=n;i++) {
    D[i] = INF;
}
D[1] = 0;

s.insert(make_pair(0,1));
/// in s tinem perechi (cost, nod) doar pentru nodurile
/// nealese inca si actualizate macar o data
/// in paralel tinem distantele minime la noduri si in D

/// perechile din ste sunt (D[nod], nod)

while (!s.empty()) { /// un nod se poate extrage doar o data aici,
deci
                     /// acest while nu poate avea mai mult de n pasi
    p = s.begin()->second;
    s.erase( s.begin() );
    // extragem in o(1) minimul (inlocuieste forul de aflare a minimului)
    for (i=0;i<L[p].size();i++) {
        vecin = L[p][i].first;
        cost = L[p][i].second;
        if (D[ vecin ] > D[p] + cost) {
        // daca se actualizeaza D al unui vecin, in cazul in care el este in set
        // (daca anterior mai fusese actualizat) stergem ferechea formata cu vecniul
            cost si nod
        // apoi adaugam parechea cu costul actualizat si nod
            s.erase( make_pair(D[ vecin ], vecin) );
            D[ vecin ] = D[p] + cost;
            s.insert( make_pair(D[ vecin ], vecin) );
            // T[vecin] = p;
        }
    }
}
```

Concluzii:

Nu am scris și cod pentru determinarea drumului efectiv. Principiu este cel clasic, acela de a păstra mereu pentru un nod tatal său ca fiind cel din care i-am actualizat minimul prin muchie directă (am atins asta prin linia comentată la Dijkstra: `T[vecin] = p`).

Roy Floyd permite determinarea drumurilor minime între oricare două noduri, pe când ceilalți doi algortitmi între un nod dat și toate celelalte.

La Roy Floyd avem nevoie și de mai multă memorie, fiind necesar să stocăm graful cu matrice de costuri, spre deosebire de ceilalți unde folosim liste de vecini. De asemenea și timpul de calcul este mai mare.

Spre deosebire de Dijkstra, Bellman Ford permite determinarea drumurilor minime și pentru grafuri cu costuri negative pe muchii. Totodată permite să detectăm existența ciclilor de cost negativ.

Diferența principală dintre Bellman Ford și BFS este că Bellman Ford un nod poate ajunge în coadă de mai multe ori.

Algoritmul Dijkstra este cel mai bun în general, însă de multe ori Bellman Ford este o alternativă excelentă.

Probleme de fixare:
https://www.infoarena.ro/problema/royfloyd
https://www.infoarena.ro/problema/rf
https://www.infoarena.ro/problema/rfinv
https://www.infoarena.ro/problema/risc
https://infoarena.ro/problema/bellmanford
https://infoarena.ro/problema/lanterna
https://www.infoarena.ro/problema/dijkstra
https://www.pbinfo.ro/probleme/591/firma
https://www.infoarena.ro/problema/catun
https://www.infoarena.ro/problema/trilant

Bibliografie

- https://infoarena.ro/
- https://www.geeksforgeeks.org/dsa/dijkstras-shortest-path-algorithm-greedy-algo-7/
- https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php